

Wollok – Relearning How To Teach Object-Oriented Programming

Nicolás Passerini^{*†}, Carlos Lombardi^{*}, Javier Fernandes^{*} and Pablo Tesone[‡]

^{*}Universidad Nacional de Quilmes

[†]Universidad Nacional de San Martín

[‡]INRIA – Francia

Abstract—Students often have difficulties in learning how to program in an object-oriented style. One of the causes of this problem is that object-oriented languages require the programmer to be familiarized with a big amount of non-trivial concepts. For several years we have been teaching introductory OOP courses using an *incremental learning path*, which starts with a simplified OOP model consisting only of objects, messages and references. This learning path is supported by a customized development environment which enables the creation of programs using this *simplified programming model*, and allows us to postpone the introduction of more abstract concepts like classes or inheritance.

In this work we propose an enhancement to this learning path focusing on the transitions between the stages of the course. We also present a new educational programming language named Wollok, which allows maximizing the accuracy in the selection of concepts to present. Finally, Wollok is accompanied by a programming environment which has lots of tools to guide the student and to help detecting mistakes and at the same time is in line with the most common professional practices.

I. INTRODUCTION

Teaching how to program has revealed itself a difficult task [8], [17]. We have individualized three specific aspects present in many initial programming courses that hinder the learning process: a complex programming language, too much concepts needed for a first working program and programming environment that are not conceived for the specific needs of an initial student [33].

There have been several proposals to address the difficulties in introductory OO courses by defining a specific language which provides a simplified programming model such as Karel++ [5] and Mama [14]. This approach has been used even outside the OO world [10], [30], [22]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [16], Traffic [24] and BlueJ [4].

The great differences between these programming languages and environments show that they have to be analyzed in the light of the pedagogical approaches behind them. The tools are of little use without this pedagogical view. For example, some educational languages and environments are designed to be used in *object-first* approaches, *i.e.*, for students without any previous programming knowledge [3], [7].

Previous works from our team [20], [21], [13], [34] have described an approach consisting of (a) a novel path to

introduce OO concepts, focusing on objects, messages and polymorphism first, while delaying the introduction of classes and inheritance and (b) a reduced and graphical programming environment which supports the order in which we introduce the concepts, by allowing to build OO programs without the need of classes. Our approach focuses on the concepts of object, message, reference and object polymorphism, while delaying the introduction of more abstract concepts such as types, classes and inheritance. These way of organizing a course provides a more gentle learning curve to students and allows them to write completely working programs from the first classes.

While this approach proved to be successful in providing the students with a more profound knowledge of OOP at the same time as raising pass rates, we feel that there is still room for improvement, in four areas: (a) the difference between the experience in the classroom and the reality in (most) professional environments, both in the language as in the development tools. (b) the gap between the simplified programming model and the classical model, mostly because of the differences between the development tools (c) adherence to an industrial language limited pedagogical decisions

As a result, we decided to conceive both a programming language and an accompanying development environment, that follows closely following the pedagogical approach we advocate for an initial OOP course. The main goal of this paper is to describe Wollok¹, the tool we created that reunites language and environment. We point out the pedagogical considerations that suggested us to build Wollok, and how they influence various of its design decisions. We also report briefly our two-year experience using this tool in initial OOP courses.

In Section II we present the problems of learning Object Oriented programming, and the consequences of this difficulties to the students. Section III describes the proposed language and the design goals and ideas we take in consideration for it. In Section IV we describe the integrated development environment we have developed for Wollok and all the features it has and how they are useful for the teaching of programming skills. Section V analyses the different design decisions we have taken, while Section VI compares our solution with another similar approaches. Finally, we summarize our contributions in Section VII, along with some possible lines of further work derived from this initial ideas.

¹ <http://www.wollok.org/>. Source code and documentation can be found in Github (<https://github.com/uqbar-project/wollok>). Wollok is open-source and distributed under LGPLv3 License (<http://www.gnu.org/copyleft/lgpl.html>).

II. PROBLEM DESCRIPTION

One cause behind the difficulties in learning OOP is that courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details, rather than the underlying object-oriented modelling skills. Also, the use of industrial languages to introduce OOP requires the student to understand several concepts before being able to run his first program. [18]

By focusing on those details, many students fail to comprehend the essential model that transcends particular programming languages [1]. Figure 1 shows a typical first program, written in Java [2]. To get this program running, the student has to walk through a minefield of complex concepts: packages, classes, scoping, types, arrays and class methods among others. However, he is still unable to do more basic OO programming, such as sending a message to an object.

```
package examples;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Fig. 1. Sample initial Java program which diverts student attention from the most important concepts.

For all these reasons, courses tend to spend too much time concentrated on the mechanistic details of programming constructs, leaving too little time to become fluent on the distinctive characters of OOP, such as identifying objects and their knowledge *relationships*, assigning *responsibilities* and taking advantage of *encapsulation* and *polymorphism* to make programs more robust and extensible.

To make things worse, frequently the students do not have proper tools that could help them overcoming all the obstacles. Already in 1999, Kolling *et al.*, had established the importance of environments in introductory courses [18]. He stated that earlier introductory courses focused on the development of algorithms in procedural or functional languages. To do this, an editor and a compiler was all that was needed for the practical part of the work. Modern courses now use object-oriented languages and subject material taught includes testing, debugging and code reuse. This creates the need to deal with multiple source files and multiple program development tools from the very start. To give a beginning student a chance to cope with this increased complexity, better environment support is needed.

The failure of students to understand the essential object-oriented concepts shows both in academy as in industry. In academic environments we find very low completion rates in introductory OO courses. Moreover, students in their very beginning of an informatics career which fail their introductory programming courses, are often likely to give up their studies [25]. In industrial development, we find that these hindrances reduce the opportunity of students to apply the concepts of the paradigm effectively in their further professional practice, resulting in several IT-projects not taking advantage of the

possibilities offered by the potential of good object-oriented practices [20].

Our previous work in this area is based on the proposal to change the order of subjects in OOP introductory courses [20], starting with a *Simplified Programming Model* (SPM) which includes only the most basic OOP concepts, such as objects, messages, references, methods and dynamic dispatch. The aim of the SPM is to provide students with the minimal set of concepts that allows them to write OO programs, exploiting polymorphism right from the beginning of the course. This foundations allow for an *Incremental Learning Path*, *i.e.*, more abstract concepts can be added later, giving the student the time to be familiarized with one concept before introducing the next one.

To enable this we provided a simplified OO language with an ad-hoc educational programming environment named Ozono², in which students can create objects and define their behavior directly, without the need for classes and inheritance [13]. Ozono is based on Smalltalk³, and thus it is a *dynamic language*.

We take special advantage of the dynamic characteristic of the language because it allows very simple uses of polymorphism, *i.e.*, any two objects that understand a common set of messages can be treated polymorphically without worrying about inheritance or *interface* implementation, which is indispensable if we want them to exploit polymorphism already in their programs after only two or three lectures.

Another interesting advantage is the ability to interact with the objects directly, sending different messages to them and see how they react. This interactive sessions increases the understanding of the paradigm and the way the responsibilities are distributed in an object oriented program.

This approach produced impressive results and therefore has been adopted in other courses and universities. The simplified programming model, allows the students to spend more time working with the essential concepts of the OO paradigm. This, in turn allows for more practice and more complex exercises, even incorporating more advanced OO techniques such as *design patterns* [12], which in traditional approaches are normally postponed to a second OOP course.

Still, the experience of eight years in four universities⁴ and thousands of students has provided us with new insights of the learning process, which is what lead us to introduce several adjustments in our approach.

In the first place, starting with a *classless* language mandates a change of language and environment in the middle of the course, which is confusing for some students. To solve this, we propose to integrate *class-based* and *standalone objects* in the same language.

²The name has been changing along these years. Other names for our programming environment have been ObjectBrowser, Loop and Hoop.

³The latest version currently in use in universities is based on a Smalltalk dialect named Pharo [6].

⁴Ozono is currently used in Universidad Tecnológica Nacional, F. R. Buenos Aires and F.R. Delta and Universidad Nacional de Quilmes; and it have been used in Universidad Nacional de San Martín and Universidad Nacional del Oeste.

Second, we think that the absence of static typing information (which Ozono inherits from Smalltalk) prevents our programming environment to aid the students in finding some typical beginner mistakes. Still, we do not want to force the students to add type annotations to their code, because that would distract them from the concepts we want to focus on. We propose to settle this apparent controversy by enhancing our programming environment with a type inferer. By doing so, the student is not required to care about type annotations, and at the same time we can detect some programming mistakes and aid the student in solving them.

Last, we have detected that sometimes the students which seem to understand the main concepts and can apply them in interesting ways to create medium to complex program, then have a hard time translating this knowledge to their professional activity. We think that a good mitigation plan for this problem starts with bringing the activities in the course as close as possible to the professional practice. For that matter, we propose to incorporate industrial best practices such as code repositories and unit tests, adapted to the possibilities of students with little or no programming experience.

The renewed approach is supported with a new programming language, named Wollok, and a programming environment which aids students to write, test and run programs. Wollok is designed to give support to our pedagogical approach; it allows to define both classes and standalone objects, incorporates a basic type inferer and provides a simple syntax to define unit tests.

A big difference with Ozono and other educational OO languages is that Wollok is a *file-based* language. While we recognize the value of the *image-based* approaches, we also are aware that most industrial programming environments are file-based. Therefore, we think that a file-based approach will shorten the distance between the classroom and the professional activity, which is one of our main goals. Moreover, image-based environments tend to blur the distinction between the programming environment and the program under development, which can embody great possibilities for advanced programmers but often does not more than confuse beginners. Also being file-based allows the usage of the most popular code repositories such as Git, and also reduces the gap for integrating the language with existing tools like code-reviewing, static code analysers and code coverage tools. These tools can be easily integrated to the programming environment, giving the student the a simplify introduction to these professional tools.

Finally, the programming environment incorporates several advanced characteristics from professional environments, which improves the coding experience, and at the same time allows the student to familiarize himself with the kind of tools he will face in his later professional activity. For that matter, the environment incorporates content assist, automatic refactorings, advanced code navigation, language semantics-aware search tools and error highlighting *while-you-type*

While neither the language itself nor the programming environment contain novel features that are unseen in industrial tools, the assemblage of selected features, each one carefully selected due to its educational value, is not found in other

previous programming environments neither educational nor industrial. Therefore, the distinctive characteristic of these tools is the search for a programming toolset which (a) supports our pedagogical approach, (b) feeds the student with a set of tools which are adequate to his current knowledge and (c) gently prepares him to be using industrial-level tools constitutes a novel way of dealing with the problems of OOP teaching.

A big amount of effort in our research has been put in looking for solutions that can solve the apparent controversy between the objectives (b) and (c). Often, the rich set of tools an industrial language or programming environment offers, cannot be exploited by an unexperienced programmer or even confuses him. On the other end, we think that poor programming environments fail to help the students to make his first steps in programming and so trims the possibilities of introductory courses. Therefore, there is much to gain from a language that has the exact features a teacher wants to teach and a programming environment which provides the exact tools a student can take advantage off at each time of his learning process.

The current study and development has been focused on university students which have had a previous subject on imperative programming. The natural extension of this work is the adaptation of these ideas to teenagers or more generally students without any prior programming experience.

III. WOLLOK: THE LANGUAGE

Wollok is a brand new language, built to specifically give support to our pedagogical approach. Many ideas have been inherited from our previous projects, such as Ozono or Loop. The most important of these inherited characteristics is the ability to create objects and treat them polymorphically without the need of type annotations, classes or inheritance. Also, as its predecessors and unlike other pedagogical tools, Wollok is a *general purpose* language, *i.e.*, it is not tied to any specific domain.

One of the main objectives of building a new language is to provide a smoother transition from the first phase of the course, in which students use a *simplified programming model* (SPM) and the second phase, in which they use a full-fledged OOP language. With our previous programming tools students were required to discard, in the middle of the course, the programming model and environment they already knew and were used to. This transition has sometimes been traumatic, because the process to define an object has to be re-learned and the tools they had been using up to that point are no longer available. With our new approach, the tools they first learned are going to continue being available through all the course, together with the new ones that are incorporated in later lectures. This is also consistent with some modern industrial OO languages that allow to define both classes or standalone objects, such as Scala [28].

Also we reduced to a minimum the syntax and the most basic constructs of the language. While this objective was already present in our previous work, the implementation strategy of Wollok allows us to go much further in accomplishing this goal, *cf.* Sec. ???. The example in figure 2 shows how

the classical hello world would look in Wollok. To build this first program students are not required to know about typing, scoping or packaging. The only required construct is the `program` and the only command is a message send. Both the receiver and the parameters are built-in objects which will be handled in the same way as user-defined objects. The concepts required to understand this program are no more than program, object, message and argument passing.

```
program {
  console.println("Hello World!")
}
```

Fig. 2. Sample initial Wollok program.

Another concept we propose to emphasize in the first programming courses is the control of side effects, *i.e.*, a programmer should be aware of the potential side effects of each portion of code. The most basic feature in Wollok to control side effects is the ability to differentiate variables (defined using `var`) from constants (defined using `val`). One step forward is to incorporate an *effect system* [27], *cf.* Sec. VII.

To sum up, there are several simple features which help structure the way a student sees his programming activity. *Object literals* and *collection literals* reduce boilerplate on object creation, since we think that the excess of bureaucracy to create an object helps to build up the belief that using objects or collections is far more complicated than using numbers or strings, which in turns leads to *primitive obsession* [11].

Each Wollok file has to be defined as *program*, *library* or *test*. Only programs and tests can be run. Libraries can only be *imported* from programs, tests or other libraries. This concepts push students onto modularizing their programs into smaller units that can be reutilized.

Figure 3 shows some of the mentioned features. The programs includes two objects which are treated polymorphically, collections and block closures. Students should be able to build such a program after four lectures. In the first lecture we introduce objects, messages, methods and references; in the second one we focus on polymorphism; and in the following two we work with blocks and collections.

IV. A CUSTOMIZED PROGRAMMING ENVIRONMENT

Beginner programmers are likely to require more guidance and make more mistakes than experienced programmers. Therefore, we think that is much to gain from a good programming environment which structures the programming experience and helps the students to identify common mistakes.

The Wollok programming environment includes a lot of features that provide guidance to the student. *Content assist* shows the students what are his possibilities at any moment and feeds automatically into the code the most usual constructs, allowing the student to concentrate less on syntax and more in the modelling of the exercise problem. *Quick-fixes* allow Wollok not only to highlight problems in the student's code but also to propose automatic solutions for some usual mistakes.

```
program myProgram {
  val optimistic = object {
    method hiThere() {
      "Hi, what an amazing day !"
    }
  }

  val pessimistic = object {
    method hiThere() {
      "Don't talk to me, it's a terrible day!"
    }
  }

  val all = #[optimistic, pessimistic]

  console.println(all.map[p| p.hiThere()])
}
```

Fig. 3. Simple polymorphism example.

Advanced code navigation and *smart reference searches* allow the programmer to better understand the dependencies in his program. Moreover, *automatic class diagrams* provide a high level view of the program and also helps understanding.

Unlike many traditional OO programming environments, which are image-based, Wollok is file-based. While we have found solid grounds for taking this decision (*cf.* Section ??), we also recognize the importance of a *live object environment*, *i.e.*, a work space in which the programmer can interact with live objects by sending them messages. As in many file-based OO and scripting languages, in Wollok this kind of interaction is achieved through an interactive console or REPL⁵ The interactive console allows the programmer to inspect the state of his program or modify it, both at the end of an execution or in the middle of a debug session. However, right now we do not provide a way to modify the program while it is running, as it happens in classical image-based environments. This kind of features have been postponed because we think that modifying the code in the middle of a program run usually has subtle consequences that are difficult to grasp for an unexperienced programmer. It is not uncommon to see that students get confused when they try to modify live code, so there is a high price to pay with little to gain in return.

A. Detecting mistakes

The programming environment has many tools intended to help detecting mistakes. We make special emphasis in detecting errors *while* the student is writing code. *Syntax highlighting* helps identify the most simple mistakes by providing immediate feedback when something is not right. Moreover, the environment provides *real-time highlights* for several kinds of mistakes (*cf.* Figure 4). Finally, the *type inferer* allows to detect more subtle mistakes.

This validations are organized and shown in a unified way, using a dedicated section of the user interface for their display. All the results of the checking and the validation of the program is shown in one integrated view, it is called *Problems View*, Fig. Figure 5 shows a view of this feature.

⁵Read-eval-print-loop [15].


```

4 var pepita = object {
5   var energy = 0
6
7   method fly(distance) {
8     energy -= distance
9     this.toRest()
10  }
11 }

```

Fig. 4. Detection of an error sending a message to *this* which doesn't exist

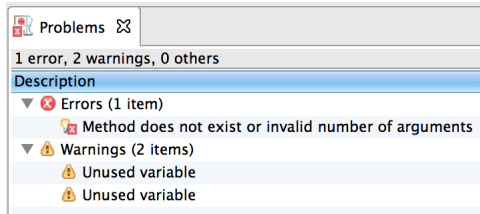


Fig. 5. Problems View: shows the different problems detected by the IDE

Checks and validations are not only used to show type errors or syntax errors, but also to encourage some properties of the program we consider as main topics in the learning process of an OO language. Here is a list of all the validations and checks the tool supports, and a brief reason why they are useful while teaching object oriented programming:

- **Syntax Errors:** this category involves all the errors detected by the parser and the lexical analyser of the language.
- **Style Errors:** this category is useful to teach good practices and to start to talk about code quality, reuse and code sharing.
 - *Case in Names:* respecting the difference case conventions for names (e.g., using *camelCase* starting with lower case for variables, using camel case starting with upper case for classes).
 - *Order and grouping:* inside the definition of an object or class the internal references are declared first, then the constructors and finally the methods.
 - *Modularization:* the classes can only be defined in a library and not in the main program.
 - *Duplicated references:* it is impossible to declare a reference using an already used name. This encourages the idea of avoiding name shadowing and improves the readability of the program.
- **References resolution problems:** these errors are useful to detect and avoid references to undeclared variables and also errors in the sending of messages.
 - *Undeclared references:* from local variables, parameters or internal fields of objects and classes.
 - *Undefined constructors:* checking for the number and type of the parameters.
 - *Messages to this:* sending messages to *this* is a special case, here we can check the existence of the correct method by the number and type of the arguments, even without using type inference.

- **Reference usage:** these errors are useful for the detection of erroneous or *dead code*, such as unused variables or references, sending messages to never assigned variables, using variables instead of values, existence of the overridden method. Figure 6 shows an example of an unused variable error.

- **Type Errors:** the errors are useful for the validation of the compatibility between the references, its possible types, and the messages sent to them, this is performed by the type system and its inferer (e.g., message sending, assignation of variables).

```

4 var pepita = object {
5   var energy = 0
6
7   method fly(distance) {
8     energy -= distance
9     this.toRest()
10  }
11 }

```

Fig. 6. Detection of unused variables

B. Type Inference

Another distinctive characteristic of the Wollok project is the type inferer. We think that type inference is key to a simple programming environment. On one side, it allows to detect lots of common mistakes *before running the program*: if an object does not understand a message, if a wrong argument is passed, if incompatible types are mixed or even miss-spellings. In environments without this capability it takes more time to detect errors. Moreover, it is not uncommon that a type mistake produces a runtime error in a place different from where the mistake was done, producing confusion.

Still, providing a type inferer for a language such as Wollok has many subtleties, which deserves an independent study [29]. On one side we require it to be able to work without type annotations and at the same time provide feedback useful for an inexperienced programmer. On the other side, the type system is rather complex; for example, the presence of stand-alone objects requires the type system to handle *structural types*, since a named type system would not allow them to be treated polymorphically. Also, we want to be able to treat polymorphically stand alone objects with class-based objects. Figure 7 shows an error detected by the type inferer and how it shows the information to the programmer. Notice that the inferred type for the object *ufo* is a structural type: fly

```

var ufo = object { // a bird !
  method fly() { 'moving my wings' }
  method eat() { 'eating...' }
}

ufo.fly()

ufo = object { // superman
  method fly() { 'fist ahead, flying !' }
  method burnWithLaserEyes() { 'Burning !' }
}

ufo.eat()

```

Fig. 7. Type system in action, detecting not defined method for the message sent

C. Beyond showing mistakes

The IDE is not restricted to showing what is wrong, but also generates proposals known as *quick fixes*. Figure 8 shows an example of one of such proposals. In this case the IDE detects that we are sending a message that the receiver does not understand and proposes to create the corresponding method.

```
var pepita = object {  
  var energy = 0  
  
  method fly(distance) {  
    energy -= distance  
    this.trest()  
  }  
}
```

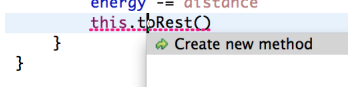


Fig. 8. Quick fix tool for common errors and mistakes

All these tools allow the student to gain more control of his code, keeping him away from feeling lost, which is otherwise a common situation for a student walking his first steps into programming. In this way the IDE becomes useful in the objective of teaching programming concepts, instead of only showing syntax errors.

V. DISCUSSION

A. A brand new language

A common point of controversy is whether it is worth to create a brand new language and toolset, instead of building our pedagogical ideas on top of existing ones, such as Self, Ruby, Smalltalk or even Eiffel. In our experience, beginning programmers require different features from their working environment than advanced programmers and the right selection of tools and concepts can produce substantial improvements in the learning process. Therefore, we believe that the possibility of fine tuning provided by a specialized environment largely pays for the additional effort.

Each semester, a group of more than 20 teachers in 3 different universities share their experience with the language and tools and discuss about new features and changes to the system. Every modification is guided by a shared understanding about how to teach OOP [20], [21], [13], [34]

A good example about teaching-specific language-design decisions is Wollok import system, *i.e.*, the way that a programming language allows the programmer to refer in one unit of code (for example a file) to program entities defined elsewhere. The import system allows the student to write his first very simple programs without knowing about packages or modularization, which are far too complex for him at the beginning. Still, later in the course modularization concepts are introduced and even the language forces the student to separate his code in different units. A full description of how the import system works and other syntax decisions can be found in [?].

B. To IDE or not to IDE

Another frequent controversy between software programmers is about the convenience of using an IDE or a simpler text editor for writing code. In the last decade, several languages, frameworks and other tools have become popular for which there are fewer visual or integrated environments.

This scarcity of tools has diverse roots. In some cases, the lack of type information undermines the possibility to implement features such as code completion, automatic refactorings or code navigation. In other cases the velocity of change in languages and frameworks makes it impossible for the tools to catch up. Frequently there is also a matter of taste, some (maybe younger) developers prefer lighter programming environments. In the teaching environment, it has been claimed that providing the student with too much tools will make them dependent of those tools.

In our view, tools that simplify day to day work can not be neglected. We recognize that the availability of tools for several modern technologies is limited, but still we see that professional programmers make use of a good amount of tools to program consistently and efficiently. Proof of this is that the most popular text editors in industry are those that allow for additions in the form of plugins, where the programmer can create his own personalized development environment. Other tools that are not integrated into the development environment, are inserted into the development process by other means; for example a continuous integration process may run a *linter* on each commit, check the build and run tests.

So, instead of a discussion about whether we need powerful tools, we rather see an evolution from heavy monolithic environments with lots of tools onto an ecosystem of light tools that have different ways to integrate with each other allowing a developer or team to create a unique environment which accommodates to their specific needs and taste.

Still, in our specific case, we opted for an *integrated* environment, because it simplifies the set up for beginners as they only have to install one piece of software which comes with all the tools they will need for the course. In more advanced courses, we think that it could be a good idea to let the students build their own environments.

Finally, we think that teaching programming should include teaching the best practices that we see in the professional world. A student which knows the best practices and tools that are used in professional software development will have a significant advantage over those who lack these knowledge.

VI. RELATED WORK

The first aspect to analyse is the shape of OOP introductory courses. Vilmer *et al.*, [35] presents a work exposing the advantages of the implementation of object-first introductory courses. Also, Moritz *et al.*, [26] presents a way of starting the learning of a programming language using an object-first way using multimedia and intelligent tutoring. Another interesting work in this area is the one from Sajaniemi *et al.*, [32] who presents another way to introduce the main concepts. All this authors propose to use an industrial language, such as Java or C#, but they do not address the problems arising from the use of these languages. On the other hand, Lopez *et al.*, [22] present a successful way of teaching using functional-first in an introductory course.

Another aspect to analyse is the use of an industrial programming language or a custom one. In this subject, the approach of Lopez *et al.*, [22] is similar to ours, but in a functional-first solution. As Wollok, his language is focused on

the main concepts of the paradigm. Another custom language specifically built to focus on the main concepts of OOP is BlueJ [4]. This implementation shares with Wollok the idea to simplify the language, but it is class centered. Wollok is both class and object centered, so we avoid need to teach classes to start learning the basic concepts of the paradigm.

There are interesting works in the Visual languages as a way of teaching OOP: Scratch [23], Etoys [19] and Kodu [31]. Still, all of them are far away of a professional development environment, so the transition to a industrial level work is not so easy as with Wollok.

VII. CONCLUSION

Wollok is an educative, object-oriented programming language which is accompanied by an advanced programming environment. Both tools are highly customized to give support to an introductory OOP course. Our approach consists of the combination of these three cornerstones: incremental learning path, customized language and specialized programming environment.

First, we defined an *incremental learning path* choosing exactly which are the concepts we want to teach and the order we want to teach them. The learning path starts with a *simplified programming model* (SPM), i.e., one which uses less concepts than a full-fledged OOP language. The SPM allows the student to build simple programs without requiring more advanced concepts. The path should attach the SPM with a good set of programming exercises, specifically oriented to be easy to build in the selected SPM. Once the student has mastered the concepts on the SPM, we can go a step further and introduce the next set of concepts.

Next, defining our own programming language, allows us to give full support to the selected learning path, avoiding the need of explaining complex concepts too soon in the course or forcing the student to write *boilerplate code* which he cannot yet understand. Finally, a good programming environment, helps detecting errors, provides guidance and most significantly allows the student to *explore*. We have found that often students are afraid to search for solutions not seen in the class or test their own ideas, which leads them to restricting themselves into a smaller set of concepts and tools they feel more secure about. A controlled environment empowers students to look around and explore new possibilities.

One major objective in our future work is the integration of more *automatic user interaction* tools into the Wollok environment. Our objective is to enable the students to have visual and interactive programs without requiring them to learn the subtleties of GUI building, extending the ideas in Gobstones [22] to object-oriented domains. Some advances in this area can be seen in our previous work named Hoope [9].

The second major objective is to continue improving the detection of programming errors. A cornerstone to achieve this goal is the type inferer, which is our current focus. The other half of our future work in this area is a powerfull *effect system* [27].

Another characteristic of programming in the real world is the need to work in teams. The success of object-oriented

languages is partly due to their advantages in group projects. It is necessary teach our students about the techniques needed for teamwork, right from the beginning. To do this, it is essential that the environment has some form of support for group work [18]. Therefore, we plan to create simplified tools to integrate wollok te *version control systems*.

Also we are working in adding more automatic refactor tools, and a better type inference implementation. Even working on adding an effect system to detect correct usage of the language and the code conventions.

One of the important development steps to be done is the implementation of a web version or a lighter version, using less hardware requirements, with the aim to run the solution in small netbooks like the ones in the *Conectar-Igualdad* program⁶.

Finally, in the educational use of the tool, we will be testing it in different educational environments to get feedback about the learning experience; generating learning material (e.g., examples, exercises, guides). As the focus of the tool is to provide a new way of teaching programming skills. For this objective, we will be working in collaboration with Universities, Teachers and non profit organizations.

ACKNOWLEDGEMENTS

We want to thank all the people who participated in the ObjectBrowser, Loop, Hoope and Ozono projects, as well as the teachers and students that provided feedback from their use of those tools, leading us to the ideas presented here.

REFERENCES

- [1] Computing curricula 2001. *J. Educ. Resour. Comput.*, 1(3es), Sept. 2001.
- [2] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 1996.
- [3] D. M. Arnow and G. Weiss. *Introduction to programming using java: an object-oriented approach*. Addison Wesley, 1998.
- [4] 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.
- [5] J. Bergin, J. Roberts, R. Pattis, and M. Stehlik. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1996.
- [6] A. P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Pharo by Example (Version 2010-02-01)*. Square Bracket Associates, 2010.
- [7] K. B. Bruce, A. Danyluk, and T. Murtagh. A library to support a graphics-based object-first approach to CS 1. In *ACM SIGCSE Bulletin*, volume 33, pages 6–10. ACM, 2001.
- [8] E. Dijkstra. On the cruelty of really teaching computer science. *Communications of The ACM*, 1989.
- [9] Estefania Miguel, Miguel Carboni, and Nicolás Passerini. Hoope, construyendo un lenguaje para enseñar, Nov. 2013.
- [10] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon. Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*, 4(2):13–17, Apr. 1970.
- [11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [12] E. Gamma and K. Beck. *Contributing to Eclipse*. Addison Wesley, 2003.

⁶<http://www.conectarigualdad.gob.ar/>

- [13] 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.
- [14] A. Harrison Pierce. Mama-an educational 3d programming language.
- [15] T. Hey and G. Pápay. *The Computing Universe: A Journey through a Revolution*. Cambridge University Press, 2014.
- [16] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, Oct. 1997.
- [17] 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.
- [18] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOOP*, 11(8):8–15, 1999.
- [19] Y.-J. Lee. Empowering teachers to create educational software: A constructivist approach utilizing etoys, pair programming and cognitive apprenticeship. *Comput. Educ.*, 56(2):527–538, Feb. 2011.
- [20] C. Lombardi, N. Passerini, and L. Cesario. Instances and classes in the introduction of object oriented programming. DC - FCEN - UBA, Buenos Aires, Argentina, Dec. 2007.
- [21] Lombardi, Carlos and Passerini, Nicolás. Alumnos, docentes y recorridos en una materia de programación informática. UNQ – Bernal, Buenos Aires, Argentina, Oct. 2008.
- [22] P. E. M. López, E. A. Bonelli, and F. A. Sawady. El nombre verdadero de la programación. Aug. 2012.
- [23] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. *SIGCSE Bull.*, 39(1):223–227, Mar. 2007.
- [24] 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.
- [25] I. Milne and G. Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies*, 7(1):55–66, Mar. 2002.
- [26] S. H. Moritz, F. Wei, S. M. Parvez, and G. D. Blank. From objects-first to design-first with multimedia and intelligent tutoring. *SIGCSE Bull.*, 37(3):99–103, June 2005.
- [27] F. Nielson and H. R. Nielson. Type and effect systems. In *ACM Computing Surveys*, pages 114–136. Springer-Verlag, 1999.
- [28] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.
- [29] Passerini, Nicolás, Tesone, Pablo, and Ducasse, Stephane. An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types. Cambridge, England, Aug. 2014.
- [30] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.
- [31] M. Research. Kodu.
- [32] J. Sajaniemi and C. Hu. Teaching programming: Going beyond “objects first”.
- [33] M. T. Singh. How to teach programming languages to novice student and problems in learning of students. *Journal of Computing Technologies (JCT)*, 1(2):5, 2012.
- [34] Spigariol, Lucas and Passerini, Nicolás. Enseñando a programar en la orientación a objetos. UTN FRC, Córdoba, Argentina, Nov. 2013.
- [35] T. Vilner, E. Zur, and J. Gal-Ezer. Fundamental concepts of cs1: Procedural vs. object oriented paradigm - a case study. *SIGCSE Bull.*, 39(3):171–175, June 2007.