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 programm 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 Wollok[1], 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.

Nico ►*No sé si hablar de los temas de colecciones, son un poco particulares de Ozono. También del environment se podrían decir más cosas que sólo el sistema de tipos.*◄ Pablo ►*Yo no hablaria de nada de eso.*◄

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.

[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).

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.

II. Why Wollok?

One cause behind the difficulties in learning OOP is the use of industrial languages, which require the student to understand several concepts before being able to run his first program [18]. Figure 1 shows an example of a possible 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, printing to standard output and class methods before being able to have a first object and send a message to it.

```
package examples;

public class Accumulator {
  private int total = 0;

  public int getCurrentTotal() { return total; }
  public void add(amount) { total += amount; }

  public static void main(String[] args) {
    Accumulator accum = new Accumulator();
    accum.add(2);
    accum.add(5);
    accum.add(8);
    System.out.println(accum.getCurrentTotal());
  }
}
```

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

Courses tend to spend too much time concentrated on the details of programming constructs of a specific language, leaving too little time to become fluent on the distinctive characters of OOP.

Moreover, frequently the students do not have proper tools that could help them overcoming all the obstacles. This might not be a problem for introductory other introductory courses, focused in on the development of algorithms in procedural or functional languages, but has a significative importance for object-oriented courses that want to deal with larger programs in multiple files, teaching concepts such as testing, debugging and code reuse[18].

Nico ►*Contar algo de otras propuestas anteriores a la nuestra*◄ Pablo ►*No se si pondria algo, ya estan en los trabajos anteriores que estan citando. Podemos decir que lean el tuyo con carlono.*◄

Nico ►*A lo que sigue hasta el final de la seeción le falta una pasada.*◄

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.

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 teacher 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 inperative 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. The Wollok Language

Wollok is a brand new language, built to specifically give support to our pedagogical approach. Wollok provides aextremely simple programming model, which allows the students to create programs that including objects, messages and polymorphism, but without the need for more abstract concepts such as classes, inheritance or type annotations. Later in the course, Wollok allows for the incremental introduction of more abstract concepts, providing a smooth transition into a full-fledged OO programming model.

The example in figure 2 shows an example first program in a Wollok-based OO introductory course. Syntax has been reduced to a minimum and the basic constructs of the language match exactly the concepts we want to transmit, *e.g.,* var is used to define variables and method is used to define methods. The accumulator object is defined as a stand-alone (*i.e.,* it has no visible class), automatically instantiated and *well-known* (*i.e.,* globally accesible) object (WKO).

```
object accumulator {
  var total = 0

  method getCurrentTotal() { return total }
  method add(amount) { total += amount }
}
```

Fig. 2. Sample initial Wollok program.

We put special emphasis in avoiding input and output (I/O). Normally, writing to standard output as it is shown in Fig. 1 will be considered a problem in industrial software construction. Therefore, teaching our students to try out their programs in this way is introducing a bad practice that will have to be *unlearned* later. On the other hand, we need some kind of user interaction to be able to see the behavior of our programs and proper handling of user interaction is way beyond the scope of an initial OO course.

The first tool we introduce to try out a program without requiring I/O is a *read-eval-print-loop* (REPL). Running a program in the REPL brings all defined objects to live and allows the user to interact with them sending messages, as shown in Fig. fig:repl. The REPL handles all I/O and the student is only required to write the desired *domain object*.

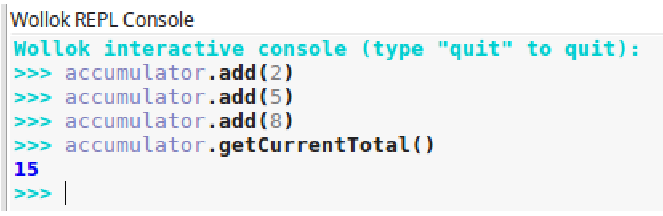

Fig. 3. Sample usage of the accumulator program in the REPL

Shortly after in the course, we introduce unit testing which slowly replace the REPL as the main form of interacting with objects. Figure 4 shows a sample test for the previous program.

```
import accumulator.*

test "adding 2+5+8 should give 15" {
  accumulator.add(2)
  accumulator.add(5)
  accumulator.add(8)
  assert.equals(15, accumulator.getCurrentTotal())
}

test "accumulator starts with 0" {
  assert.equals(0, accumulator.getCurrentTotal())
}
```

Fig. 4. Sample test Wollok program.

Again here Wollok is fine-tuned for our pedagogical objectives. For example, tests have to be written in a separate file, which introduces the student into some basic knowledge of modularization. Also, the test runner guarantees that there is no interaction between tests: the messages sent to the accumulator in the first test will not affect the state of the accumulator in the second test **Nico** ►*sería lindo tener una cita a alguna regla de unit testing que proponga esto*◄ .

Another simple feature that is very helpful in the initial steps of the course is the presence of literals for lists (*e.g.,* [1,2,3]) and sets (*e.g.,* #{1,2,3}). This allows us to increase the complexity of examples that we can build without introducing classes. For example, our accumulator object can be implemented as shown in Fig. 5.

```
object listBasedAccumulator {
  var history = []

  method getCurrentTotal() {
    return history.sum()
  }
  method add(amount) { history.add(amount) }
}
```

Fig. 5. Another accumulator implementation, using a list.

Next in the course, we introduce classes. Once more, Wollok helps us in the transition: any pre-existent stand-alone object can be converted into a class by just changing the keyword object for class[2], as shown in Fig. 6.

```
class ListBasedAccumulator {
  var history = []

  method getCurrentTotal() {
    return history.sum()
  }
  method add(amount) { history.add(amount) }
}
```

Fig. 6. A third accumulator implementation, class based.

Figure 7 shows how the three previous definitions of accumulator can be used polymorphically. Also it shows the usage of more advanced list messages and closures[3]. A special mention has to be made about the fact of stand-alone objects being used in the same program and even being polymorphic with class-based objects.

Nico ►*Esto requeriría una mención en la intro, o en general una explicación más detallada.*◄ 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 const). One step forward is to incorporate an *effect system* [27], *cf.* Sec. VII.

[2] As a matter of fact, we will also change the name, as our code convention mandates lowercase names for objects and uppercase names for classes

[3] In our approach, we introduce polymorphism and closures *before* classes. For reasons of space we skipped those intermediate examples here.

```
import accumulator.*

test "adding 2+5+8 should give 15" {
  const accumulators = [
    accumulator,
    listBasedAccumulator,
    new ListBasedAccumulator()
  ]

  accumulators.forEach { accum =>
    accum.add(2)
    accum.add(5)
    accum.add(8)
    assert.equals(15, accum.getCurrentTotal())
  }
}
```

Fig. 7. Simple polymorphism example.

IV. A Customized Programming Environment

The features that influence the experience of the learning programmer are not limited to the programming language she uses. The *tools* through with the code is written, analyzed and evaluated have a paramount relevance for this experience, and therefore for the success of the programming courses.

Beginner programmers are likely to require more guidance and make more mistakes than experienced programmers. Also, the kind of support required by experienced programmer from her development environment is different from that required by a beginner, *e.g.,* an experienced programmer might select her programming environment thinking on increasing productivity. One very important feature a beginner requires from her programming environment is *discoverability*, *i.e.,* the tools should help discover possible paths of action and gently provide feedback when the student makes a mistake, helping her to understand what was wrong and how to fix her program. Finally, all programmers require tools that help them understand, navigate and explore their programs.

We decided to embed the Wollok language in an integrated programming environment, whose features are designed having in mind the specific needs of novice programmers. In our view. the tools provided by the environment are a fundamental part of the Wollok proposal, in equal terms with the language features.

In particular, the Wollok environment provides tools focused to the following goals:

- To guide and ease the acual code writing.
- To detect several of the most common mistakes done by novices, providing adequate feedback, and even to provide possible corrections when they can be computed.
- Navigate an give different perspectives of the defined objects and classes.
- Organize the code that is produced along the course.
- Test and experiment with objects, both those provided by the student and those provided by Wollok.

We remark that several of the tools that the Wollok environment provides are common, in exact or approximate form, to those provided by mainstream industrial IDEs like e.g. Eclipse, Visual Studio or the Idea series. In this way, we aim to make both the programming experience more appealing to the students, and the transition to later courses and work environments softer; while giving adequate support to the learning process through the same tools.

A. Basic guidance for writing code

We have noticed that the first barrier for novices is the strictness of a programming language syntax. Frequently initial students find annoying that their program will not execute if they forget a closing brace or mispell a keyword or a variable name, in many cases even a case error stops execution or produces unexpected results. Syntax highlighting is very helpful by providing a very fast visual feedback about a mispelled keyword. Also, like many modern code editors, the Wollok IDE automatically creates a matching closing symbol each time the programmer types a parenthesis, square braket or braces. Also the IDE helps correctly indenting the code inside code portions enclosed in braces or square brakets, which both helps avoiding frustrating syntax problems and starts to induce best practices about code organization.

Other approaches have addressed this problem using block-based or visual programming tools Nico ►*Cita a alguno*◄ . While we value those ideas, we think that mainstream programming is and will continue to be text-based, therefore, non-textual programming can be useful for younger students, but at universitary level it is better to provide tools that help dealing with syntax problems rather than continue circumventing them.

At a bigger scale, Wollok admits the definition of several objects and/or classes in the same file. This allows the teacher to go deeper in the initial examples in playing with several objects and polymorphism, without requiring the students to struggle with imports and packages. These concepts will arise later in the course, when students' programs increase their complexity to a level that demands for modularization (*cf.* **??**). Nico ►*¿Esto no es una característica del lenguaje?*◄

The Wollok IDE also has integrated a powerful *code analyser* that enables for *content assistance* Nico ►*¿Está bien este nombre?*◄ , *i.e.,* in certain contexts, the IDE can autocomplete an identifier name, or provide a list of possible completions if there are many (*cf.* Fig. 8). This is available for all types of variables and constants and messages sent to self, super or any WKO[4].

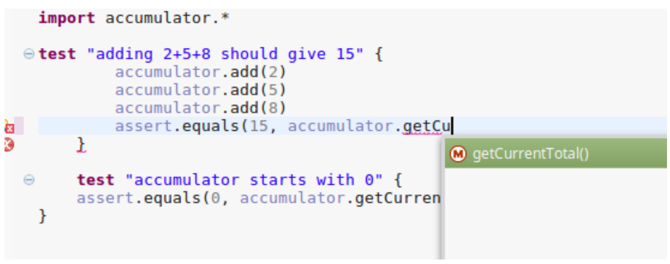

Fig. 8. A list of suggestions from the Wollok IDE content assistance.

[4]Current work in progress includes a type inference system that once completed should allow to add content asistance for any message send

B. Detect mistakes and help fixing them

Another dimension of the programming environment is helping the programmer to recover from his mistakes.

First, we aim for *static error detection* whenever possible, because it provides faster feedback than runtime checks. Some errors can be even reported as the programmer is typing. On the other hand, runtime checks are not only slower but also open to be skipped in some program executions.

Second, error reporting should provide with clear messages, explained in the same terms in which the teacher talks to the students. Also, inline error reports, *e.g.,* underlining the offending code, relieves the student from the task of mapping the detected problem with the possible cause

Third, in some situations, the IDE can propose possible automatic fixes to the detected problems. Figure 9 exemplifies these features. The mispelled identifier is underlined and if we put the mouse on the error report we get an error message and a possible fix. In this case, the automatic fix will insert an empty method in the adecquate position.

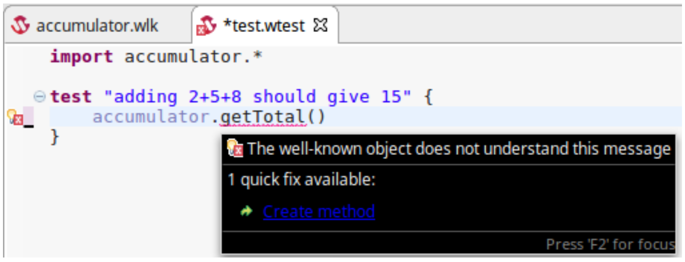

Fig. 9. Example of a simple error detection.

Together, this characteristics are meant to empower the student to explore different ideas, providing positive feedback in case he makes a mistake.

While this does not replace the more personalized feedback a teacher can provide, in several situations a sensitive automatic feedback can help the student not to stay stuck with simple errors waiting for a response. This in turn allows for more exercises during the course.

Moreover, automatic detection acts as a (basic) assistant teacher, *i.e.,* some simple topics that are not crucial for the course can be left for the student to learn by herself in the interaction with the IDE. This is specially efficient for some kind of errors that occur only for specific groups of students, *e.g.,* those with previous non-academic OO experience. Tackling this errors in class, would imply to show a bad solution to the rest of the students, that had otherwise not thought about it. This would be a fine strategy in an advanced course, but will confuse beginners. Instead, we prefer let the IDE detect the mistake and show possible solutions, only for the students that effectively incurr in these kind of errors. Nico ►*Faltaría un ejemplo.*◄

Los errores que analizamos salen de la experiencia en el aula.

To detect several of the most common mistakes done by novices, providing adequate feedback, and even to provide possible corrections when they can be computed.

- Error indications with adequate messages. De esto daría algunos ejemplos que nos parezca piola resaltar, no sería exhaustivo. - Quick fixes.

The programming environment has many tools intended to help detecting mistakes. We make special emphasis in detecting errors *while* the student is writing code. Nico ►*Acá se podría hablar más*◄ *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 10). Finally, the environment has a (basic) *type inferer* which in some cases detects more subtle mistakes.

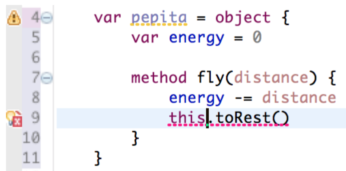

Fig. 10. Detection of an error sending a message to *this* which doesn't exist

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 11 shows a view of this feature.

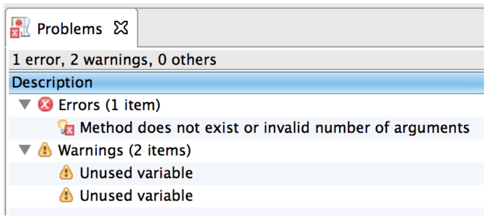

Fig. 11. Problems View: shows the different problems detected by the IDE

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**: this 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 12 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).

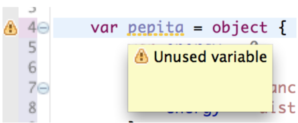

Fig. 12. Detection of unused variables

C. 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 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 13 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

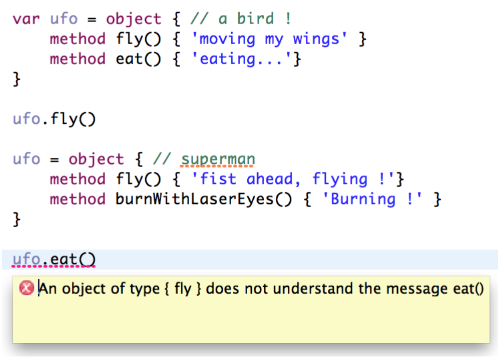

Fig. 13. Type system in action, detecting not defined method for the message sent

D. Beyond showing mistakes

The IDE is not restricted to showing what is wrong, but also generates proposals known as *quick fixes*. Figure 14 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.

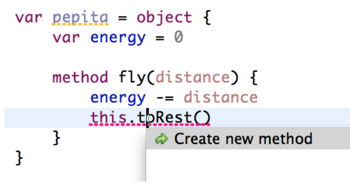

Fig. 14. Quick fix tool for common errors and mistakes

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 is 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, begginning programmers require different features from their working environment that 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 became 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 tecnologies 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 accomodates 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 approcah 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* **Nico** ►*Revisar la idea de Javi*◄ *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. **Nico** ►*Acá se podría hablar de constructivismo.*◄

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 subleties 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* program[5].

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.

[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.

[5] http://www.conectarigualdad.gob.ar/

[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.