Wollok – Relearning How To Teach Object-Oriented Programming

Nicolás Passerini*†, Carlos Lombardi*, Javier Fernandes*, Pablo Tesone‡ and Fernando Dodino§*†
*Universidad Nacional de Quilmes
†Universidad Nacional de San Martín
§Universidad Tecnológica Nacional – Facultad Regional Buenos Aires
‡IMT Lille Douai – 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.**

In this work we describe Wollok, which encompasses both an educative language and a specialized integrated development environment (IDE) conceived for learning OOP and supporting our pedagogical approach. Equally important, we describe our teaching experience with these tools and the motivations for their design.

I. Introduction

Teaching how to program has revealed itself a difficult task [6], [13]. 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 [27].

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++ [4] and Mama [11]. This approach has been used even outside the OO world [8], [24], [18]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [12], Traffic [20] and BlueJ [3].

The great differences between these programming languages and environments show that they have to be analysed in the light of the pedagogical approaches behind them. The tools are of little use without their respective 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 [2], [5].

Previous works from our team [16], [17], [10], [28] 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 three 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.

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 VI analyses the different design decisions we have taken, while Section VII compares our solution with another similar approaches. Finally, we summarize our contributions in Section VIII, 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

[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).

several concepts before being able to run his first program [14]. Figure 1 shows an example of a possible first program, written in Java [1]. 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 characteristics 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 other introductory courses, focused on the development of algorithms in procedural or functional languages, but has a significative importance for object-oriented courses where we want to deal with larger programs in multiple files, and teaching concepts such as testing, debugging and code reuse [14].

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, 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 incorporate industrial best practices such as code repositories and unit tests. Adapting them 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; it also 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 our solution 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. This approach 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 inexperienced programmer or even worst they confuse the inexperienced student. On the other end, we think that poor programming environments fail to help the students to make his first steps in programming and so trimming the possibilities of introductory courses. Therefore, there is much to gain from a language that has the exact features a teacher desires to teach and a programming environment which provides the exact tools a student can take advantage of 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. The Wollok Language

Wollok is a brand new language, built to specifically give support to our pedagogical approach [16], [17]. Wollok provides an extremely simple programming model, which allows the students to create programs containing objects, messages and polymorphism, without the need for more abstract concepts such as classes, inheritance or type annotations. Later in the course, Wollok allows 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
  var evens = 0

  method getCurrentTotal() { return total }
  method add(amount) {
    total += amount
    if (amount % 2 == 0) { evens += 1 }
  }
  method evenCount() { return evens }
}
```

Fig. 2. Sample initial Wollok object definition.

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 the 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 behaviour of our programs. However, 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. 3. The REPL handles all I/O and the student is only required to write the desired messages to a *domain object*.

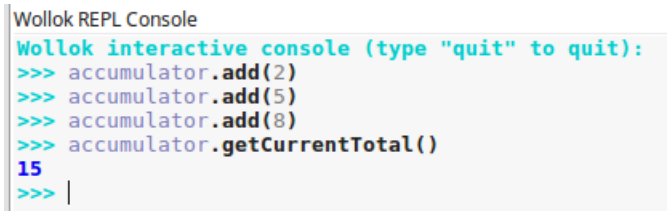

Fig. 3. Sample usage of the accumulator object in the REPL

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, Wollok is fine-tuned for our pedagogical objectives. In particular, 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.

Furthermore, tests have to be written in a separate file. However, this decision introduces a problem. The objects (and later on, classes) defined in a different file must be accesible from the test file. To this end, Wollok includes a simple form of the **`import`** directive, that refers to a object/class definition file in the same folder than the test file. In this way, some basic knowledge of modularization is subtly introduced early in the course[2].

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 use collections, and therefore increase the complexity of examples that we can build before introducing classes. We even briefly introduce *closures* at the initial stage of the course. 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) }
  method evenCount() {
    return history.count({n => n % 2 == 0})
  }
}
```

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[3], as shown in Fig. 6.

```
class ListBasedAccumulator {
  var history = []

  method getCurrentTotal() {
    return history.sum()
  }
  method add(amount) { history.add(amount) }
  method evenCount() {
    return history.count({n => n % 2 == 0})
  }
}
```

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[4]. A special mention has to be made about the fact that stand-alone objects are used in the same program and even they are polymorphic with class-based objects.

```
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.

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

[2]The assimilation of the `import` directive, as early as in the second week, was not problematic in our experience using Wollok for first-year programming courses.

[3]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

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

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). Moreover, methods not including a `return` expression are considered as void, i.e. they do not yield a value and no operation can be applied to them. Furthermore, the REPL do not show any text as result of the evaluation of void methods, as can be seen in Fig. 3.

IV. A Customized Programming Environment

The features that influence the experience of the learning programmer are not limited to the programming language used. The *tools* used to write, analyse and evaluate the code 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 actual code writing.
- To detect several of the most common mistakes done by novices, providing adequate feedback, and even to provide possible corrections whenever is possible.
- 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 bracket or brace. Also the IDE helps correctly indenting the code inside code portions enclosed in braces or square brackets, 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 . 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. Allowing the teacher to go deeper in the initial examples using 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.

The Wollok IDE also provides *content assist*, *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[5].

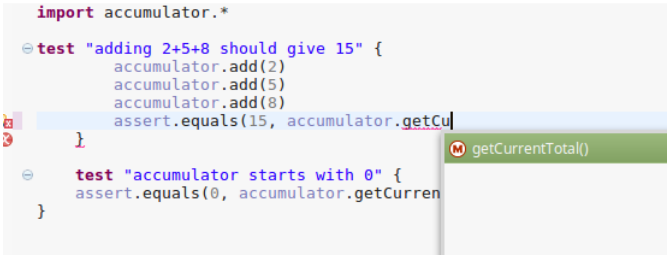

Fig. 8. A list of suggestions from the Wollok IDE content assistance.

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.

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

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.

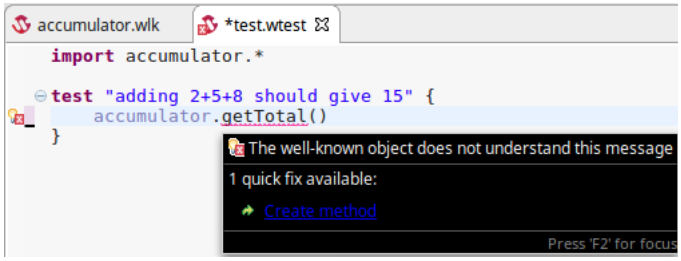

Fig. 9. Example of a simple error detection.

Together, these 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 provides, in several situations a sensitive automatic feedback helps 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.

Finally, the great majority of validations and automatic proposals, are the result of our experience in the classroom, looking at the students using the Wollok language and IDE, as well as other languages and tools we have used in the past. Each semester, we receive reports of the teachers using Wollok which include the type of errors the students make most often, trying to improve the network of validations and proposals, in order to improve the learning experience.

C. Understanding and navigating a program

Simple *code navigation* tools can significantly improve the programming experience. For example the Wollok IDE allows to go in one click from a message send to its implementation[6] and a keyboard shortcut[7] to go back to the previous location. Fast navigation back and forth through a program allows for better understanding of the relationship between the different portions of it. We have seen that lack of proper navigation tools might mislead the student to avoid correct modularization of their program, as they run into difficulties coping with a program that is divided in several small pieces.

Also, code visualization tools help understanding bigger programs and are as useful for beginners as they are for experienced programmers. The Wollok IDE provides an outline view and automatic static diagrams (*cf.* Fig. 10). These higher-level views of the program, allow the student to gain practice in abstracting herself of the details of some portions of a program, understanding her objects and classes as black boxes that provide some services, instead of attempting to have all of them in her head at all times. This level of abstraction is a necessary skill for being able to participate in a larger project.

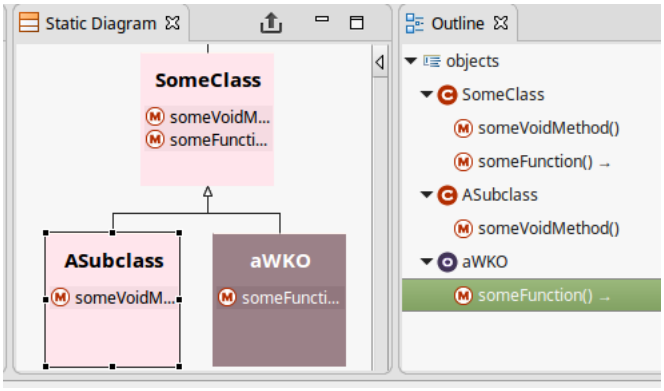

Fig. 10. Static diagram and outline.

D. Encouraging best programming practices

In our opinion, good practics should be taught from the very beginning of programming curricula. This claim is independent of the recurrent debate about rules and conventions. Our experience shows that it is unlikely that novice programmers appreciate the advantages of e.g. good variable names or correct code indentation, as these attributes are more easily appreciated on larger programs. Therefore, we have to actively enforce them, and get them used to writing and reading good quality code.

Several features of the language and code editor aim to promote, or enforce, good programming practices. For example, it compels the programmer to a predefined order inside a class definition, *i.e.,* grouping all variables and constants in the first place, then constructors and finally methods. Also it establishes some basic rules about naming and restricts the usage of global variables. We also mention that autocompletion of braces and square brackets eases the adoption of adequate indentation.

Moreover, some bad programming habits are detected by the error detection of the IDE. As an example, Fig. Figure 11 shows an example of an unused variable error. Unused, *dead code* tends to appear after a refactoring or after trying different solutions to a problem. Quick detection of this kind of errors

[6] In the cases that is identifiable by the static analyser

[7] ALT + left arrow which is familiar to students due to its universal adoption in web browsers.

will help the student to identify the bad practice and train her to avoid its appearance.

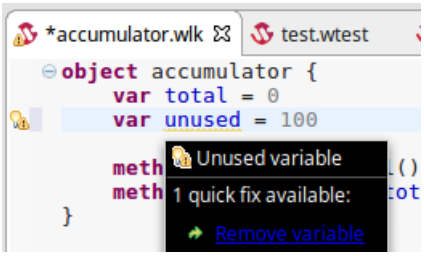

Fig. 11. Detection of unused variables

At a higher level, the Wollok IDE is designed to help the students to put in order the code they produce along the course. Source files are organized in *projects*, that have a predefined directory structure including both files for class/object definitions and tests. The generated file structure is adequate to the use of a source code repository such as GIT or SVN, improving collaboration in group assignments and serving as a communication tool with teachers.

These features solve the frequent difficulties novice programmers find to organize their source files, and in particular, to coordinate work in group assignments. We note to this respect that it is normal to see members of a work group sharing code by sending e-mail with *zip files* between each other. The time spent trying to reconcile different versions of the program or even trying to understand which is the latest one, distracts them from learning the actual topics of the course. Hence the benefit of providing suitable source organization tools, besides the promotion of good development practices it implies.

E. Tests and experiments

As we described in Section III, Wollok provides two ways of working with the defined objects and classes: the REPL and the definition of tests.

The REPL provides a simple environment for direct object manipulation; it is the first tool to interact with objects that we introduce in the course. The programmer can just send messages to the WKOs she defines, and see how they respond. In some courses, we even *start* on the REPL, sending messages to objects provided by the teacher. In this way, students get familiarized with the most important concepts of the OO paradigm: *object* and *message*, before going into the details about how these objects have been implemented.

Moreover, the REPL also allows the programmer to define local variables, which are useful to remember intermediate results to be used in further operations, making it easy for the students to perform non-trivial object manipulations.

As the REPL interaction grows, very soon the students themselves realize that they are doing repetitive operations there and start looking for automation; at this moment, *automatic tests* are introduced. Fig. 12 shows the test runner tool output for the test shown in Fig. 4.

It is important to notice that the test requires a *higher level of abstraction* than the REPL. Now the student has to anticipate the result that some expression should yield. Writing explicitly both the expression and the expected result, and interpreting a *green bar* as a signal that the answer yielded was the expected, without ever seeing it.

We observe that while the REPL is available all along the course, once the students become more fluent with automated tests, they use the REPL with less frequence. Also, we remark that by the combination of REPL and tests, we have succeeded in completely avoiding the need for undesired debugging practices, such as the inclusion of println expressions along the code.

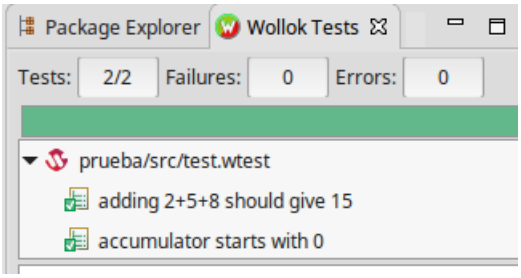

Fig. 12. Test runner view after a sucessful test run.

V. Classroom Experience with Wollok

Since March 2015 Wollok is being used on the Universidad Nacional de San Martín. In 2016 the Universidad Nacional de Quilmes and Universidad Tecnológica Nacional also started using Wollok in their introductory courses. In 2017 Wollok started being used also in a highschool.

All together, it has been used in more than 15 introductory OOP courses and by more thant 500 students. Also, many of the ideas in Wollok where present in other tools we had previously developed, going back to 2006. In these 12 years, our ideas, although always evolving, have been applied by more than 100 teachers and 6000 students. In several cases, approval rates changed from 25%–30% to 80%-90%.

Let us describe the first experience of a new student with Wollok. At the very beginning, the student starts by interacting with just two windows: the code editor, that points to an initial source file, and the REPL. All the coding is done inside the editor, that includes the features of highlighting, autocompletion, and error detection/correction we described in Section IV-B. The streamlined WKO syntax of Wollok allows to complete the definition of simple objects with a minimum of elements. A menu option allows to access to the REPL, where the defined WKOs can be accesed by their global names, so that it is straightforward to interact with them. In the interaction through the REPL, void methods are easily distinguished from those returning a value. Furthermore, typing just an object name results in a simplified internal view, that displays the attributes along with their values.

Further on, the language and IDE features permit a gentle introduction to each successive concept we work with in the course. In particular, the absence of explicit type information allows for a simple implementation of polymorphism between WKOs: it suffices to have different objects that define methods for the same message. We also remark the similarity of the syntax for WKOs and classes, described in Section Y, consent

a smooth transition to the latter, motivated by the natural desire of having multiple object that share the same definition, having at the same time separate states.

While introducing design patterns is not a part of the initial OOP subject, we observe that some patterns appear naturally as we evolve to more complex problem statements. E.g. the idea of Singleton is natural in Wollok as classes and globally accesible WKOs can be combined in the same program. Moreover, short class/object definitions can be easily added to an existing Wollok program, leading to the creation of little objects that provide specific behaviours. We note that the features of Wollok favor the reification of concepts that are not linked with the state that has to be modeled or handled. Sometimes the resulting objects follow, e.g., the Strategy or State [9] patterns.

VI. 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 [16], [17], [10], [28].

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 [7].

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.

VII. Related Work

The first aspect to analyse is the shape of OOP introductory courses. Vilmer *et al.,* [29] presents a work exposing the advantages of the implementation of object-first introductory courses. Also, Moritz *et al.,* [21] 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.,* [26] 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.,* [18] 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.,* [18] 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 [3]. 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 [19], Etoys [15] and Kodu [25]. 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.

VIII. Conclusion

The Wollok language and IDE have been put into practice for already two years, targeting hundreds of students. They have been successful in supporting an incremental learning path, allowing the users to train their OO modelling skills using a very simple programming model and providing a smooth transition to more complex models.

The IDE allows students to program in a controlled environment which helps them not getting stuck, shows best programming practices and empowers them to use their intuition and test their ideas. 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.

Finally, the IDE provides customized versions of several industry-like tools, helping the students in getting familiarized with the kind of programming environment they will find in actual professional jobs. In our experience, good students often do not automatically become efficient professionals because they encounter difficulties in translating their academical knowledge into their professional practice. Letting them work with industry-like tools helps making this transition easier.

IX. Future Work

The main focuses of attention for the Wollok IDE development team are the detection of programming errors and bad practices, and the provision of quick fixes, content assistance and refactorings. A cornerstone to achieve these goals is the type inferer, which is one of our current main objectives. Still, providing a type inferer for a language such as Wollok has many subtleties, which deserves an independent study [23]. The other half of our future work in this area is a powerfull *effect system* [22].

Also, we intend to implement several improvements to the REPL, which we expect to have great impact in the programming experience. In the first place we want to propagate several of the features of the basic editor to the REPL, such as content assistance. Then, we would like to enable code modifications while a program is running in the REPL. After a reload, the REPL should re-execute all the previous expressions in the REPL and display the new results (*cf.* Scala Worksheets[8]). Finally, we would like an automatic conversion from a REPL session to suite of tests.

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 [14]. Therefore, we plan to create simplified tools to integrate wollok with *version control systems*.

Also there are some initiatives to build web-based or lighter versions of the IDE and the interpreter. This will allow Wollok to be used in context where the availability of powerful computers is restricted. There exists a (limited) Wollok web editor integrated to the Mumuki[9] platform, which yet has not been fully tested with students.

[8] https://github.com/scala-ide/scala-worksheet

[9] http://mumuki.io/

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] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 1996.

[2] D. M. Arnow and G. Weiss. *Introduction to programming using java: an object-oriented approach*. Addison Wesley, 1998.

[3] 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.

[4] 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.

[5] 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.

[6] E. Dijkstra. On the cruelty of really teaching computer science. *Communications of The ACM*, 1989.

[7] J. Fernandes, N. Passerini, and P. Tesone. Wollok – relearning how to teach object-oriented programming. In *Workshop de Ingeniería de Software y Tecnologías Informáticas*, Universidad Tecnológica Nacional, Buenos Aires, Argentina, 2014.

[8] 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.

[9] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[10] 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.

[11] A. Harrison Pierce. Mama-an educational 3d programming language.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] Lombardi, Carlos and Passerini, Nicolás. Alumnos, docentes y recorridos en una materia de programación informática. UNQ – Bernal, Buenos Aires, Argentina, Oct. 2008.

[18] P. E. M. López, E. A. Bonelli, and F. A. Sawady. El nombre verdadero de la programación. Aug. 2012.

[19] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. *SIGCSE Bull.*, 39(1):223–227, Mar. 2007.

[20] 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.

[21] 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.

[22] F. Nielson and H. R. Nielson. Type and effect systems. In *ACM Computing Surveys*, pages 114–136. Springer-Verlag, 1999.

[23] 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.

[24] 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.

[25] M. Research. Kodu.

[26] J. Sajaniemi and C. Hu. Teaching programming: Going beyond "objects first".

[27] 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.

[28] Spigariol, Lucas and Passerini, Nicolás. Enseñando a programar en la orientación a objetos. UTN FRC, Córdoba, Argentina, Nov. 2013.

[29] 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.