Wollok: language + IDE for a gentle and industry-aware introduction to OOP

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

***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 our experience, we observed that the use of programming languages and tools that differ greatly from those used in the IT industry weakens student interest, and also hampers the application of the learned concepts and techniques in subsequent labor experiences.

In this work we describe Wollok, which encompasses both an educative language and a specialized integrated development environment (IDE) conceived for learning OOP in a way that supports our pedagogical approach, and facilitates at the same time the transition to industrial environments. 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 [7], [28]. We have individualized three specific aspects present in many initial programming courses that hinder the learning process: a complex programming language, too many 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++ [13] and Mama [11]. This approach has been used even outside the OO world [30], [26], [21]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [4], Traffic [23] and BlueJ [12].

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 [3], [16].

Previous works from our team [2], [1], [10], [20] have described an approach consisting of (a) a novel path to introduce OO concepts focusing first on objects, messages and polymorphism 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. This 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 among the development tools (c) the 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 closely follow 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 led us to build Wollok, and how they influence its design decisions. We also report briefly our two-year experience using this tool in initial OOP courses.

In recent years, several pedagogical programming environments have arised, with diverse purposes. Some of them make use of block-based or visual programming, such as Scratch [5], Etoys [19] and Kodu [25]. In our vision, these tools are suitable for stimulating interest in programming and for being used in secondary education, and not beyond that stage. Other approaches, such as Gobstones [21], focus on a first universitary programming course. Finally, there are other pedagogical programming tool proposals that share our interest in a first OOP course, such as BlueJ [12], Karel++ [13], Mama [11] and Loop [10].

On the other hand, other educators propose to use industrial languages in introductory courses, such as Java [17], Eiffel [22], [23], Smalltalk [6] and Self [29].

[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).

Wollok combines some characteristics that are typically present in academical environments with others that are more easily seen in industrial ones, aiming to enrich them to fulfill the specific needs of novice programmers and the proper development of a first OOP course. Its most noticeable characteristics are: (1) the possibility of using self-defined objects to support an introduction to the topic that postpones the introduction of the concept of class, (2) the possibility of combining self-defined with class-based code objects in the same program, (3) the decission of offering an IDE with edition and code management capabilities that are fine-tuned for unexperienced programmers and (4) that both the language as the IDE share similarities with their industrial counterparts, in order to soften the later transition to the professional tools.

In Section II we present the problems of learning Object Oriented programming, and the consequences of this difficulties to the students. Section IV describes the proposed language and design goals. In Section V 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 VIII analyses the different design decisions we have taken. Finally, we summarize our contributions in Section IX, 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 [15]. 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 to overcome 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 it has a significative importance for object-oriented courses where we want to deal with larger programs in multiple files and to teach concepts such as testing, debugging and code reuse [18].

Last, we have detected that sometimes the students who 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 hand, we think that poor programming environments fail to help students to make their first steps in programming, which in turn trims 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 have 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. Methodology

Nuestra metodología de validación se compone de dos partes. En una primera etapa realizamos un análisis cualitativo, basado en entrevistas a estudiantes y docentes, con un conjunto de preguntas semiabiertas.

Entre los docentes entrevistaron se incluyeron tanto docentes que habían utilizado la metodología propuesta como otros que no. En todos los casos, los docentes tienen alguna experiencia con la metoodología tienen experiencia previa con otros métodos, lo que nos permite también realizar preguntas que comparen los resultados en ambos casos. Cabe aclarar que a pesar de que existe suficiente documentación sobre la propuesta metodológica, cada docente realiza una implementación propia, por lo que no todos los cursos se llevan acabo exactamente de la misma manera.

En el caso de los estudiantes, se entrevistó a estudiantes que estaban a punto de terminar un curso que seguía la metodología propuesta y se los consultó sobre la experiencia en este curso, comparada con otras materias, las dificultades que encontraron en cada caso y las posibilidades de aplicar estos conocimientos que han tenido en otros ámbitos. También nos interesa medir el resultado final; en este sentido nos interesa una visión teórico-práctica, definida por la capacidad de la *aplicación consciente* del contenido teórico, es decir, debe ser capaz de resolver problemas concretos pudiendo defende r desde su comprensión de la teoría la solución que propone. Las entrevistas fueron realizadas en todos los casos por docentes que no participaban de cada curso en cuestión y garantizando la anonimicidad de las respuestas, para permitir que tuvieran total libertad para expresar su visión. Asimismo se buscó formar una selección de estudiantes que fuera representativa de cada curso en cuanto a las calificaciones finales obtenidas.

El objetivo de la segunda etapa es realizar un análisis cuantitativo que permita validar o refutar las primeras conclusiones del análisis cualitativo.

A partir del análisis de las respuestas obtenidas en la etapa anterior, se formuló un cuestionario para estudiantes, que luego fue distribuidos en una decena de cursos en 3 universidades nacionales, obteniendo unas 140 respuestas.

Adicionalmente, se recolectaron datos de retención y aprobación de diferentes cursos de programación.

IV. The Wollok Language

Wollok is a brand new languages built to specifically give support to our pedagogical approach [2], [1]. 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).

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

```
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.

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 life 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*.

```
Wollok REPL Console
Wollok interactive console (type "quit" to quit):
>>> accumulator.add(2)
>>> accumulator.add(5)
>>> accumulator.add(8)
>>> accumulator.getCurrentTotal()
15
>>>
```

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

Shortly after in the course, we introduce unit testing which slowly replaces 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. With this purpose, Wollok includes a simple form of the **`import`** directive, which 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.

Finally, Wollok includes features that allow for discussing about *controlling side effects* in an early stage of programming courses, making programmers aware of the potential side effects of each portion of code. The most basic of these features is the ability to differentiate variables from constants (defined using var and const resp.). 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 does not show any text as result of the evaluation of void methods, as can be seen in Fig. 3.

[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.

```
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.

V. 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 ones. Also, the kind of support required by a 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 the 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.
- To navigate and give different perspectives of the defined objects and classes.
- To 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 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 syntactic strictness of programming languages imposes a harsh barrier on novice programmers. Errors due to mispelled keywords or lack of proper delimiter (brace/bracket/parenthesis) match are both frequent and frustrating to them[5]. The Wollok IDE code editor offers several basic features that help to mitigate such frustrations. We mention syntax highlighting, automatic insertion of the closing delimiter when the opening one is typed, and the proposal of a proper indentation scheme. We remark that the latter feature also aims to improve readability of code, and also to induce good code organization practices.

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, constants and messages sent to self, super or any WKO[6].

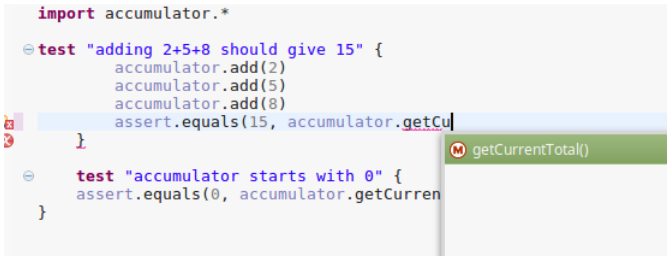

Fig. 8. A list of suggestions from the Wollok IDE content assistance.

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 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.

B. Detect mistakes and help fixing them

The Wollok IDE is able to detect statically (i.e. prior to evaluation) several mistakes frequently made by students. Apart from basic errors like non-matching delimiters, mispelled keywords and references to undefined identifiers, the code analysis performed by Wollok verifies the following:

1) Class definitions must follow a definite order: instance variable declarations, followed by constructors, followed by methods. This aims to promote good code organization practices in students.
2) Unused or never-assigned variables are indicated as warnings. The ability to separate warnings from errors opens the way to add more checks of possible bad smells in student code.
3) Similarly, a variable read just once is marked as a warning, since it could be inlined.
4) Some cases of type error related to message sends are detected. This includes all self-sends and messages sent to WKOs.
5) Some subtler mistakes are also detected, e.g. a method must return in either all execution branches or none, an abstract class cannot be instantiated.

These errors are, in fact, detected while the student is typing a program and are shown in the editor. Like in several modern code editors, the source line is marked with an error sign. When the student passes the mouse pointer over that sign, an explanatory message is shown. Special care has been put in the messages, to explain mistakes in the same terms in which the teacher talks to students. Finally, in some situations, the IDE proposes possible *automatic fixes* to the detected problems.

Figure 9 exemplifies this feature along with the rendering of a code error. The mispelled identifier is underlined and if we pass over the mouse on the error report, we get an error message together with possible fix. In this case, the automatic fix will insert an empty method in the adecquate position.

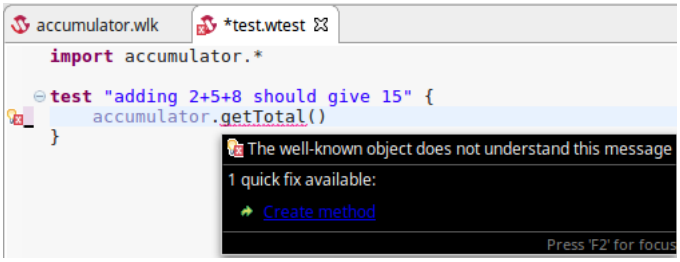

Fig. 9. Example of a simple error detection.

These features aim to give a quick visualization and a proper understanding of coding mistakes. In this way, we intend to empower students to explore different ideas, providing positive feedback in case of mistakes. While this does not replace the more personalized feedback a teacher provides, in several situations, a sensitive automatic feedback helps students not to stay stuck with simple errors waiting for teacher or colleague assistance. This, in turn, allows for more agile lab classes and, therefore, for the possibility of including more exercises during a course.

We note that 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, instead of to being explained in class. This is specially useful for some errors that only some students are likely to make. Tackling these errors in class would imply to show a bad solution to the rest of the students, who had otherwise not thought about it. While anticipating errors could be a fine strategy in an advanced course, it mostly confuse beginners. Therefore, we prefer to let the IDE detect the mistake and show possible solutions, only for the students that effectively incur in this kind of errors.

C. Understanding and navigating a program

Code navigation and visualization tools can significantly improve the programming experience. We note that in our

[5]Of course, block-based and visual programming tools make these problems just vanish. As we describe in the discussion, such tools would not be adequate for the intended uses of Wollok.

[6]Extending autocomplete to every message send is a difficult task, due to the lack of explicit type information. Current work in progress includes a *type inferer* that will improve the content assistance capabilities of Wollok.

teaching (and also industrial) experience, lack of such tools might mislead students (resp. developers) to avoid correct modularization of their program, as they run into difficulties coping with a program that is divided in several small pieces.

The Wollok IDE offers several keyboard/mouse shortcuts for code navigation, allowing e.g. to jump from a class reference (tipically, for instance creation) to its definition, from a message send to the corresponding method (when such method can be determined) and back-and-forth navigation on code portions resembling that of Web browsers. These tools allow for a better understanding of the relationship between the different portions of the code being developed by a student.

Some code visualization tools are available as well. An outline view and automatically generated static diagrams (*cf.* Fig. 10 right and left resp.) provide streamlined views of the classes and WKOs defined. Clicking a class, WKO or method pops up the corresponding definition in the code editor. We claim that these tools are particularly useful to induce students to abstract themselves from the details of some portions of a program, understanding objects and classes as black boxes that provide some services, instead of attempting to have all of them in their head at all times. This level of abstraction is a necessary skill for being able to participate in larger projects.

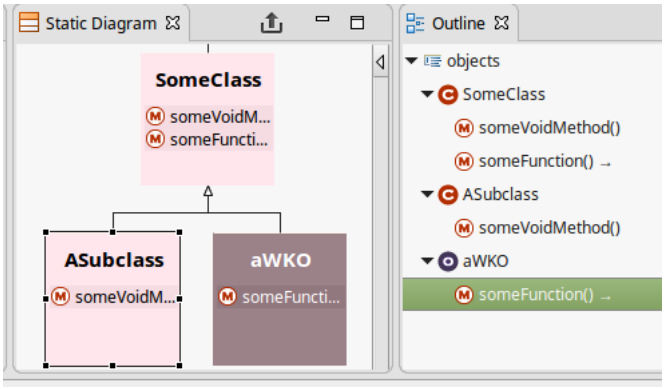

Fig. 10. Static diagram and outline.

The Wollok IDE also includes some tools to help students to manage the whole set of code produced along a course. Source files are organized in *projects*, which 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. These features alleviate 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. Another benefit of providing suitable source organization tools is the enforcement of good development practices.

D. Tests and experiments

As we described in Section IV, 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 by 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, in a short time the students themselves realize that they are doing repetitive operations there and start looking for automation; at this moment, *automatic tests* are introduced. Fig. 11 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, once the students become more fluent with automated tests, they use the REPL with less frequence, even while it is available all along the course. 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.

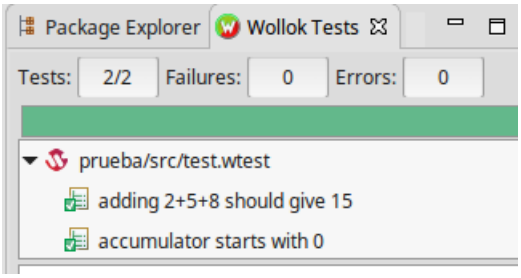

Fig. 11. Test runner view after a sucessful test run.

VI. Results

Since March 2015 Wollok is being used to teach introductory OOP courses at university level. Until 2017 it has been used in more than 30 courses in five different universities, reaching almost 1000 students. It has also been used at highschool level.

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 30–40% to 80–90%. Still, we consider that approval rates are of little value, without confirming that the knowledge level required to pass the course is at least the same as before. Indeed, the level of the courses has been consistently increased since our methodology

was implemented, covering more topics and requiring the students to build bigger programs with higher design quality.

We considered the difficulty in exams as a metric of the knowledge level acquired by studends. Before implementing our methodology, the programs required to pass an OOP exam in UTN used to consist of only one or two classes, with no more than 6–8 methods, and only one (fairly simple) usage of polymorphism. Current exams require the student to write a program with 10–15 classes and no less than 20 methods (excluding getters and setters). The number or polymorphical sets of classes/objects has been increased to 2–3, frequently including at least one that requires to come up with a non-obvious abstraction in order to sort out the problem.

In 2017 in UNQ, 88% of the students that started the course took the final exam, 96% of those where able to complete the required program (and tests) in less than four hours. The final approuval rate was 79%.

For a better understanding to the effect of using Wollok, we've realized surveys and interviews, organized according to the ideas described in Section III. We surveyed 15 courses during 2017, obtaining 133 responses. These surveys show that 84% of students found Wollok at least significantly easier to understand than other languages (*cf.* Fig. 12), and 95% thinks that Wollok has helped them understand their mistakes and learn from them. At the same time 70% of the students consider that having learnt Wollok will help them in their professional practice.

	Abs.	Sign.	Part.	Not	N/A
Easier to learn than other langs.	33	74	16	4	4
Helps understanding errors	47	78	5	1	0
Similar to prev. known langs.	19	44	31	15	22
Helps professional activity	26	52	18	15	19

Fig. 12. Student evaluation of Wollok Language and IDE. For each question, each student had to select between "Absolutely", "Significantly", "Partially" or "Not at all".

The similarity with professional practice was a major design objective of Wollok, as we consider it to be a major motivational point. Students tend to decline in attention when they fail to see the application of the concepts that are being taught. Interviews confirm that students have no problems recognizing this applicability; even more, in some cases they perceive more easily the applicability of the concepts taught with Wollok than other concepts they had learnt using *industrial* languages, such as C++. For example, several students with professional experience affirmed that learning OOP with Wollok led them to improve their professional practice. Others indicated that having learned OOP with Wollok was of significant help to learn other modern OOP languages, such as *Swift*.

These insights also allow us to confirm that the subject allows the students to get a good grasp of theoretical knowledge, that they are aware of the new theory they incorporated and that they are capable of applying these concepts in different technologies and situations. We have checked this assertion in the final path of initial courses as well as in following advanced OOP courses. For example, some courses include a final lesson in which the introduce an industrial language, such as Java. In other courses we have introduced a (extremely simple) *game-building framework* known as Wollok Game, which requires the student to use their knowledge in new (frequently more complex) ways.

VII. Classroom Experience with Wollok

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, which points to an initial source file, and the REPL. All the coding is done inside the editor, which includes the features of highlighting, autocompletion and error detection/correction we described in Section V-B. The streamlined WKO syntax of Wollok allows for defining simple objects with a minimum of elements. A menu option gives 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 IV, 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. For example 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.

VIII. Discussion

A. A brand new language

A common point of controversy is whether it is worth to create a brand new language and toolset instead of building our pedagogical ideas on top of existing ones, such as Self, Ruby, Smalltalk or even Eiffel. In our experience, beginner programmers require different features from their working environment that advanced ones. 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.

These considerations made us favor the crafting of an ad-hoc language over choosing existing ones, even those allowing to define WKOs like Self or Ruby.

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 [2], [1], [10], [20].

A good example of 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 [14].

B. To IDE or not to IDE

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

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

C. Enhancing programming practices

In our opinion, good programming practices should be taught from the very beginning of programming curricula. In fact, we claim that the teaching of programming concepts, principles and techniques should be *integrated* with that of software development practices and habits, forming a single body of knowledge. This claim is independent from 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. We remark that good practices include the right selection of development tools and the proper use of them. This consideration greatly influenced the decision of offering a complete IDE as part of Wollok.

Some of the features of the Wollok language and IDE have been conceived with the objective of promoting good coding habits. We mention the proposal of a proper indentation scheme, the signaling of dead code and other possible bad smells as warnings in the code editor, and the imposition of some degree of organization in class definitions, described in Section V. In turn, the inclusion of a simple modularization scheme in the language and the set of code navigation and visualization facilities included in the IDE promote the care for good code organization in a software project.

IX. 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 to avoid getting stuck, shows them 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 restrict 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 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 them to make this transition easier.

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 deserve an independent study [24]. Also, we plan to include an *effect system* [8].

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[7]). Finally, we would like an automatic conversion from a REPL session to a 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 to 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 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.

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] C.Lombardi and N.Passerini. Alumnos, docentes y recorridos en una materia de programación informática. UNQ – Argentina, Oct. 2008.

[2] 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.

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

[4] 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.

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

[6] S. Ducasse. *Squeak: Learn programming with robots*. Apress, 2006.

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

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

[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 Intl. 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] 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.

[13] 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.

[14] 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.

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

[16] K.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.

[17] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with java. In *ACM SIGCSE Bulletin*, volume 33, pages 33–36. ACM, 2001.

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

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

[22] B. Meyer. Towards an object-oriented curriculum. In *TOOLS (11)*, pages 585–594, 1993.

[23] 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.

[24] N.Passerini, P.Tesone, and S.Ducasse. An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types. Cambridge, England, Aug. 2014.

[25] M. Research. Kodu.

[26] R.Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. Wiley, New York, NY, USA, 1st edition, 1981.

[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] 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.

[29] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.

[30] 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.

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