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, to write even the simplest programs. 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. 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 [4], [14]. We have individualized three specific aspects present in many initial programming courses that hinder the learning process: a complex programming language, the need of introducing too many concepts to produce a first working program, and a programming environment not conceived for the specific needs of an initial student [13].

There have been several proposals to address the difficulties in introductory OO courses. Some of them define a specific language which provides a simplified programming model such as Karel++ [8] and Mama [6]; others even provide a whole programming environment specifically designed to aid novice programmers such as Squeak [3], Traffic [12] and BlueJ [7]. 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.

Previous works from our team [2], [1], [5], [11] have described a novel path to introduce OO concepts focusing first on objects, messages and polymorphism while delaying the introduction of classes and inheritance. A reduced and graphical programming environment supports this path, allowing to build OO programs without the need of classes. This environment is an adaptation of the syntax and class browser of Pharo Smalltalk. After introducing classes, the course switched to the standard Pharo development tools.

This way of organizing a course provides a more gentle learning curve to students, so that they can write completely working programs from the first course weeks.

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 and the classical programming models, mostly because of the differences among the development tools (c) the adherence to a preexistent, general-purpose language that 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.

Wollok combines some characteristics that are typically present in academical environments with others that are more easily seen in industrial ones. 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.

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 [10]. Figure 1 shows an example of a possible first program, written in Java [9]. To get this program running, the student has to walk through a minefield of complex concepts:

[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).

packages, classes, scoping, types, arrays, printing to standard output and class methods; just 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. Hence we advocate the use of a pedagogically conceived programming language that allows to build simple programs from a *minimum* of concepts, along with a programming environment specifically tailored for the needs of novice programmers.

We have also detected that sometimes, 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 bringing the activities in the course as close as possible to professional practice could help mitigate this problem. For that matter, we aim to incorporate industrial practices such as unit tests, adapting them to the possibilities of students with little or no programming experience.

Wollok is designed to give support to our pedagogical approach: it allows to define both classes and standalone objects, includes an industrial-like, simplified IDE, and provides a simple syntax to define unit tests as well as a graphical interface to run them.

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.

III. 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).

```
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 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*.

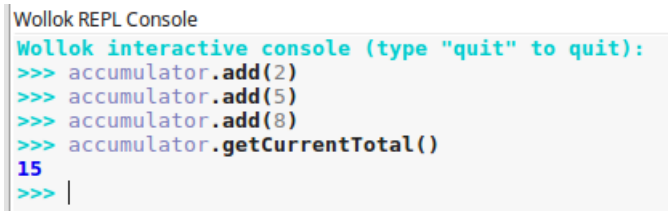

Fig. 3. Sample usage of the accumulator object in the REPL

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.

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.

[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

```
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.

```
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.

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.

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

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.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.

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] 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.